Middleware for Building Distributed Applications Infrastructure

(A State-of-the-art Report on Middleware)[*]

Rod Fatoohi[†] David McNab[‡] David Tweten[§]

NAS Technical Report NAS-97-026
December 1997

Abstract

Applications in modern enterprises are increasingly dependent on two enabling technologies: high-speed networking and object orientation. These applications, which are becoming more complex and distributed, typically run on different computers, running different operating systems and software tools, and are interconnected by different networks. Clearly, there is a need for integrating different components efficiently and reliably in a distributed heterogeneous environment. This paper presents an overview of the state-of-the-art middleware technology for building distributed applications infrastructure. Four middleware technologies are considered: the Distributed Computing Environment (DCE) by The Open Group, the Common Object Request Broker Architecture (CORBA) by the Object Management Group (OMG), the Distributed Component Object Model (DCOM) by Microsoft Corp., and Java by Sun Microsystems Inc. The architecture, implementation, and services provided for each are discussed in detail. Finally, some comparisons and concluding remarks are provided.

1. Introduction

Applications in modern enterprises increasingly depend on information technologies that rely heavily on two enabling technologies: high-speed networking and object orientation. High-speed networking provides the medium for information to move quickly between different systems while object orientation provides the extensibility, maintainability, and reusability that are needed in software engineering. These technologies are the basis for many state-of-the-art applications. These applications are becoming more complex and distributed. With many applications, information resources—as well as their users located at multiple sites—are interconnected over fast networks. Remote users can create, remove, modify, and view data using different software tools. Current and future applications include: airline reservation, on-line transaction processing, multimedia, digital library, air-traffic control simulation, and virtual manufacturing. These applications need an infrastructure support in order for data to move freely between different systems.

Most of today's organizations have a wide variety of computers, that run different operating systems and software tools, and are interconnected by different networks. A typical organization may have many personal computers running Microsoft Windows or MacOS, as well as workstations and servers running different versions of Unix and Windows NT,

[*] The work of Fatoohi and McNab was funded through NASA contract NAS 2-14303.
[†] College of Engineering, San Jose State University and MRJ at NASA Ames Research Center, email: rfatoohi@email.sjsu.edu.
[‡] MRJ.
[§] NASA Ames Research Center.

mainframes running MVS, and even supercomputers running Unix, used mainly for large database and scientific applications. These computers rely on different networks and protocols for communication. They might be connected locally by Ethernet, FDDI, and Fibre Channel, and to the outside world through ATM and Frame Relay networks. Various protocols are also used for communication, including: TCP/IP, IPX/SPX, SNA, and NetBEUI.

Applications tend to be even more diverse than the computers and networks they use. Some applications can only run on a single platform, while others can run on multiple platforms. Some can be accessed through a network while others are available only on a local server. In addition to the heterogeneity problem, there is a problem with legacy applications, which tend to be relatively large being developed many years ago and have evolved in an unstructured manner. However, these applications are mission critical and need to run successfully all the time. These applications are harder to adopt to a new platform than others. Clearly, there is an need for integrating different components in a distributed heterogeneous environment. This may require building an infrastructure to support different applications efficiently and reliably. Such an infrastructure would tie different computers over different networks and be transparent to users.

Many organizations have realized such a need and have tried to solve these problems by forming standards committees or by introducing certain products that would either become de facto standards or adopted as standards. Both the Object Management Group (OMG) and The Open Group were formed to develop standards for building a distributed applications infrastructure. At least two vendors, Microsoft Corp. and Sun Microsystems Inc., are offering products for building such an infrastructure.

This paper presents an overview of the state-of-the-art middleware technology for building distributed applications infrastructure. Four are considered: the Distributed Computing Environment (DCE) by The Open Group, the Common Object Request Broker Architecture (CORBA) by the OMG, the Distributed Component Object Model (DCOM) by Microsoft and Java by Sun. The following section provides an introduction to middleware. The middleware architecture, implementation, and services are discussed in detail. Finally, some comparisons and conclusions are given.

2. Middleware

Middleware—like many relatively new, high-level system concepts—is not a well-defined technical term. For the purpose of this study, we define middleware as a set of common services that enable applications and end users to exchange information across networks [Umar 1997]. These services *reside in the middle* above the operating system (OS) and networking software and below the distributed applications [Bernstein 1996]; as shown in Figure 1. Middleware services have several properties that distinguish them from applications or low-level services [Bernstein 1996]:

- They operate on different OS and network services. This means that they are not tied to a specific platform such as computing hardware, communication networks, operating systems, and other software components.
- They are used for different applications. This means that they are available to multiple applications and users and are not embedded in a single application for a specific industry.
- They are distributed, which means that these services can be accessed remotely or enable applications to be accessed remotely.
- They support standard interfaces and protocols. This is a very important property since it means that these services are specified by well-defined Application Programming Interfaces (APIs) and communication protocols. Such interfaces make it easier to port

applications to different platforms. Also, standard communication protocols allow applications on systems with different machine architectures and operating systems to exchange data.

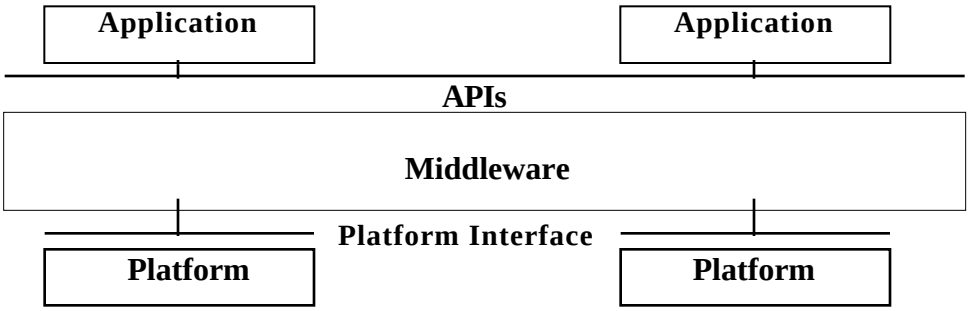

Figure 1. Middleware

In general, middleware can be divided into a client component, which runs on every client machine, and a server component, which runs on server machines. The client component is a set of software modules that can be invoked by the client applications through an API. The server component monitors the client requests and invokes appropriate server processes [Umar 1997].

Middleware starts with the API set on the client side that is used to invoke a service, and it covers the transmission of the request over the network and the resulting response [Orfali et al. 1996a]. Middleware does not include the network services that provide end-to-end transport services, such as the TCP/IP protocol stack. These services are now considered low-level or first-generation middleware services.

Specific middleware services change over time. Some services migrate into the OS if they are important and run on all popular platforms. In this case, better performance can be achieved. Also, some OS components migrate to become middleware services to simplify the OS implementation and make that component available for all platforms.

Middleware has many advantages including:

- Hides the platform software and hardware details from end users, so it releases the applications from the complexities of network protocols, the operating system, and local services.
- Makes application distribution transparent to the users. This means that the same operation can be performed locally or across a network.
- Provides portability and interoperability. This means that applications can be ported to different environments and that they interoperate.
- Enables new applications that use its services. Here, middleware can reduce the costs and risks of developing these applications.
- Provides flexibility and scalability.
- Identifies several important components that can be used and shared across many environments.
- Helps deal with legacy applications. These applications can be encapsulated as a set of functions that can be accessed by middleware services.

On the other hand, middleware has problems, including:

- *Maturity*—The middleware technology is not mature enough to be used in many mission-critical projects. There is a gap between the basic concepts and practice. For many middleware services, the specification has been developed but no real implementations or real products exist. Also, middleware is part of the infrastructure design, so if there are no products, the framework developers have to develop their own middleware services. In addition, since the technology is evolving, choosing the *wrong* middleware might cause a major problem with the project.
- *Standardization*—Many middleware services use proprietary protocols and APIs and are not based on standards. This means that many products are neither portable nor interoperable.
- *Cost*—Middleware adds to the cost of the environment. Client and server components are required for all the systems in the environment.
- *Performance*—Middleware may cause a performance degradation in bandwidth and latency, since adding a software layer increases the communication overhead if it is not fully integrated with the other layers.
- *Complexity*—Many middleware services are complex and not well understood. There are not enough trained people to manage and maintain these services.

Middleware provides many services. These services can be categorized into three broad areas [Umar 1997, Rymer 1996]:

- Communication (information-exchange) services that handle communication between the client and the server. They provide message format, message representation, message services (such as marshaling, unmarshaling, and compression), message binding, synchronous and asynchronous communications. Not all these services are provided by every middleware product. Usually, these services run on one or more network transport protocols such as TCP/IP, IPX/SPX, and SNA.
- Management and Support (core) services that provide name-management services (map logical names to their physical addresses), security (access control and privacy), failure handling (timeouts, deadlocks), performance measurements and monitoring, and memory management.
- Application-specific services that provide services for classes of applications. Among these services are SQL database access, transaction processing, and data-replication services.

Early middleware mainly provided communication services. Today's middleware has many of the services listed above. We focused on middleware that provides many services in addition to the communication services. These middleware technologies are: CORBA, DCE, DCOM, and Java.

There are various ways to categorize middleware. One is middleware with a standards committee behind it or it is mainly supported by a company with a major market share. (In our work, a consortium-based committee is considered as a standards committee.) Both CORBA and DCE are supported by standards committees, while DCOM and Java are supported by specific companies. Another category is whether the middleware is general or service-specific. All of them (CORBA, DCE, DCOM and Java) are considered to be general. However, there are many other middleware technologies that target specific markets, such as database-specific middleware; see Orfali et al [1996a] and Umar [1997] for details. Yet another way is whether the middleware is object-oriented or procedure-oriented. CORBA, DCOM, and Java are object-oriented, while DCE is procedure-oriented.

The following sections describe the four listed middleware technologies in detail, along with the services they provide. Many assessment parameters are also given for each middleware, including: maturity, availability, usability, interoperability, and unique characteristics.

3. DCE

3.1 DCE Overview

The Distributed Computing Environment (DCE) is a software infrastructure for developing distributed systems. It consists of a set of application programming interfaces and a set of run-time services, which together provide the fundamental functionality required for distributed applications. DCE is based on a simple but flexible remote procedure call (RPC) paradigm. In the middleware arena, DCE is generally considered *low level*, in that it implements primitive fundamental services [Rosenberry 1992].

DCE is produced by the Open Group, an integrator specializing in distributed systems and open systems standardization.

3.2 DCE History

3.2.1 The Open Group

The Open Group was formed in February 1996 through the consolidation of X/Open and the Open Software Foundation (OSF). X/Open had previously focused on open systems specifications and standards compliance; the OSF was a software research and development organization with distributed systems and operating systems products. The Open Group describes itself as an international consortium of vendors and end-users dedicated to the advancement of multi-vendor information systems.

The Open Group is primarily a software integrator and standards advocate. Its stated philosophy is that open systems are comprised of multiple, successful technological components, each conforming to standards, merged to form a single system. The Open Group tries to select appropriate technology, or in some cases develop it, and integrate it into a coherent information system [The Open Group 1996 & 1997].

Significant sponsors of DCE include Digital Equipment Corp., Fujitsu, Hewlett-Packard, Hitachi, IBM, NCR, Novell, Siemens Nixdorf, and Sun. The Open Group lists Bull S.A., SGI, and SCO as *associate sponsors*.

3.2.2 Source Technology

The Open Software Foundation built DCE using a technology solicitation and integration process. Requirements documents were prepared covering all major functional areas needed to provide a general-purpose distributed computing infrastructure. These requirements were written into a technology solicitation Requests For Comments. OSF then selected the best responses, according to their criteria, and the technology was integrated into the reference DCE implementation. As one would expect given this methodology, all major components of DCE were derived from extant products or research projects. For example:

- DCE threads are based on POSIX threads (draft 4).

- The DCE Security Service is based almost entirely on MIT's Kerberos.
- Cell Directory Service (CDS) is derived from X/Open's XDS.
- Global Directory Service (GDS) comes from Siemens' DIR-X ISO (CCITT X.500) system.
- Distributed File Service (DFS) is based heavily on AFS, developed jointly by IBM and Carnegie Mellon University.

The piecemeal nature of DCE is central to its character and is responsible for many of its strengths and weaknesses.

3.2.3 Availability and Interoperability

One of the strengths of DCE is the number of interoperable implementations that are available. According to The Open Group, DCE is the most widely ported distributed systems infrastructure product. Vendors providing DCE products include Bull S.A., Cray Research, Data General, Digital Equipment Corp., Fujitsu, Hewlett-Packard, Hitachi, IBM, NEC, Olivetti, Siemens Nixdorf, Sony, Silicon Graphics, Stratus, and Tandem Computers. There is an OpenVMS implementation of the secure core services, as well as Microsoft Windows (NT and 95) implementations of the client software. In addition, Windows NT implements its own RPC interface that uses the same wire protocol as DCE RPC and is interoperable. However the API is gratuitously different, so applications do not port well.

The Open Group provides a reference DCE implementation and a set of conformance tests. Resellers can either purchase the reference implementation and port it to their platform or develop a DCE product of their own. In either case, to be officially sanctioned as *DCE interoperable* the implementation must pass the Open Group's conformance tests. As a result, current DCE implementations have very few problems interoperating.

3.3 DCE Architecture

Like all RPC-based systems, DCE employs a client-server paradigm. Multithreaded servers implement procedure bodies and export their interfaces through a hierarchical global namespace, called the Cell Directory Service (CDS). Clients make remote procedure calls to named services, using the DCE Security Service to authenticate themselves and gain the necessary access privileges. CDS converts service names to the low-level information that the client needs to contact the server. The details of name-to-server location translation, procedure parameter marshalling, and network communication are entirely transparent to users.

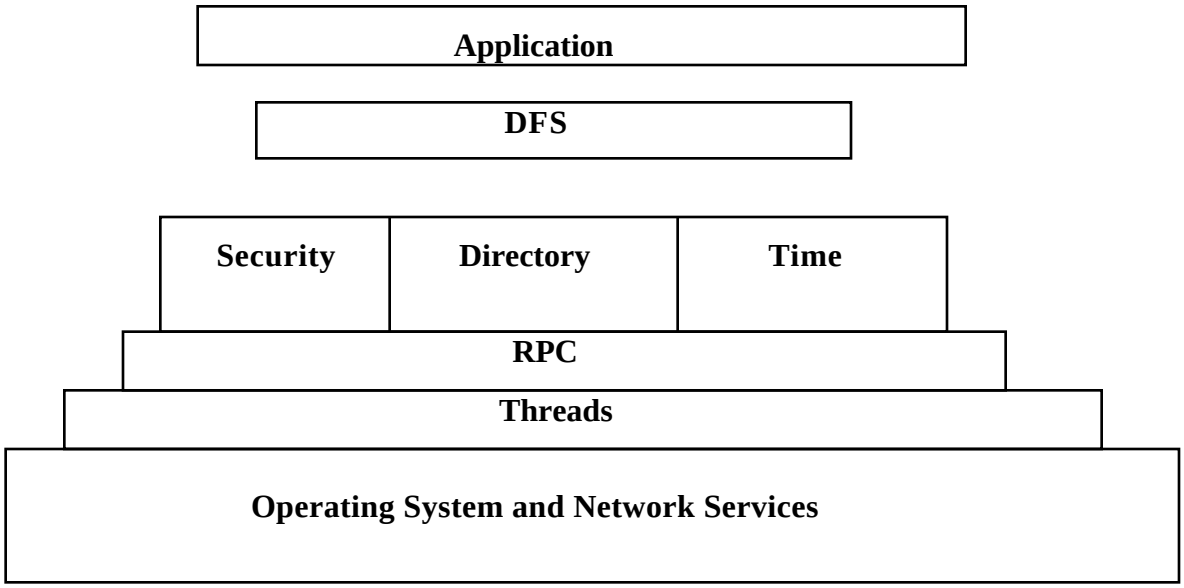

Figure 2. DCE Architecture

In implementation, DCE consists of a set of software packages layered on top of *vanilla* operating systems (see Figure 2). Core services—naming and security—are provided by server programs, typically running on dedicated hardware and typically replicated for fault tolerance. Client software consists of API libraries and system applications built on them, as well as some daemons that run locally and help manage access to DCE services. Support of DCE's core services requires no modification of the underlying operating system.

The unit of DCE deployment is a *cell*, which at minimum consists of the core servers, the machines they run on, and any client nodes. Through features of its naming service, DCE supports location hiding and redundancy. A DCE application uses a service by making an RPC to a service name, rather than to a specific (host, port) pair. This indirection allows a DCE cell to provide fault tolerance and load balancing for the core services or for user applications.

DCE is strongly based on its RPC paradigm. It is in no way an explicitly object-oriented system. As The Open Group points out, DCE does not preclude the development of object-based systems; for example, a secondary implementation of the CORBA ORB interoperability protocol uses DCE RPC. On the other hand, it provides little support. The API is entirely C-based, with no support for object-oriented languages or for inheritance. Nor does the CDS namespace support object interfaces or inheritance. Despite The Open Group's attempts to recast DCE as infrastructure for a distributed object model, the fit is clumsy at best. A reasonable analogy is the use of C for object-oriented programming—it is certainly possible, and C components can be used to build more abstract object-oriented systems, but neither C nor DCE is likely to be a programmer's first choice.

3.4 DCE Services

DCE provides a set of core services that The Open Group defines as the fundamental infrastructural components for distributed computing. These include a naming service, a

security and authentication service, and a system for synchronizing the clocks of nodes that belong to the environment. The first two are essential for higher level DCE services and are clearly fundamental to distributed computing.

3.4.1 CDS: Naming

The Cell Directory Service (CDS) provides a global namespace for DCE applications. Servers export their interfaces by assigning them a CDS name. Clients look up the service name and use the returned location information to communicate directly with the server. For example, a printing service could assign itself the CDS name "/.:/services/printing" ("/.:" indicates the local cell namespace). Associated with the namespace entry would be the IP address for the machine running the printing server, the protocol to be used to connect to it, and some ancillary information. The run-time library for the client program decodes this information and uses it to connect to an endpoint manager on the server machine. The endpoint manager provides information to guide the client to the local port number at which the server program is listening for RPC requests. Thus, the client is able to dispatch an RPC to a server knowing only the service name.

CDS is hierarchical, consisting of a tree-structured namespace of inter-linked directories. Directories contain objects of several types:

- Pointers to other directories, which provides the mechanism for building a hierarchy.
- Profiles describing services defined in the local cell (a simple mechanism for finding which services are available).
- Server objects with information necessary to allow an RPC client to bind to the server interface.
- Groups of server objects.
- The equivalent of symbolic links, for creating short-hand paths or multiple spanning trees.

Each directory and its contents can be replicated for fault tolerance and load balancing. Replicated directories are weakly consistent and in some cases must be updated by hand. Clients typically access the CDS server closest to them in the network topology and, if their request cannot be satisfied, it is directed to another server. Client-side programs called *clerks* provide caching services to improve performance.

Object groups provide a generic method for supporting fault tolerant, load-balanced services. A *server object entry* maps a service name to the binding information necessary to make an RPC to the program implementing the service. *Object groups* allow one name to map to multiple pieces of binding information, representing multiple servers. If one is down or heavily loaded, the others can transparently satisfy the client's request.

The CDS hierarchy also includes *junctions* that allow external services to resolve parts of CDS pathnames. Because a CDS pathname is parsed component by component, a junction may be encountered. The junction indicates that the remainder of the pathname has meaning only in an external namespace. In addition, the junction contains the binding information necessary to make a remote procedure call to the external namespace manager, so that it can resolve the remainder of the path. This useful feature enables subsystems with their own naming paradigms to be linked into the global CDS namespace. For example, the DFS file namespace is accessible to CDS clients via the "/.:/fs" junction point. Any reference to a CDS name prefixed with /.:/fs will be handed off to the DFS name-resolution subsystem, which will treat it like a standard DFS pathname.

3.4.2 DCE Security

The DCE Security Service is a direct descendant of Kerberos [Kohl 1989]. It uses symmetric cryptography (i.e. private key) techniques built on Data Encryption Standard (DES) algorithms to implement a credential-based, distributed, security system [Hu 1995].

The base authenticated agent in DCE security is called a *principal*, which is usually a human user but can also be a server program or a host machine. Associated with each principal is a dynamic set of credentials, called *tickets* in Kerberos nomenclature, that provide access to the services available in the DCE cell. A special-purpose credential called the Ticket Granting Ticket (TGT) is acquired when the correct account password is provided at login time. The TGT carries authorization information and is used to acquire tickets for other services as they are needed, obviating the need to repeatedly enter a password. The risk of credential stealing is mitigated by associating an expiration time with each ticket, typically about twelve hours. However credential stealing is probably the top threat for DCE security. Kerberos was designed for use on single-user workstations, and in a shared environment credential caches are vulnerable to misappropriation by privileged users [Bellovin 1991].

Central to the security service is the *registry*, a database of principals, groups of principals, accounts, and cell-wide security policies. The registry can be replicated for load balancing and fault tolerance. DCE clients provide remote access to the registry via the Security Service daemon.

Access to DCE services is controlled by an authentication protocol and a set of access control lists (ACLs) associated with each service. Recall that each authenticated agent in the DCE cell is called a principal, and these principals can be users or server programs. Associated with every principal is an encryption key known only to the principal and to the Security Service. To use a service, the client presents a *service ticket* to the target server. The service ticket contains the identity and privileges of the client, encrypted in the target service's private key. The target service decrypts the service ticket, determines the identity of the client, and uses its *ACL manager* to determine whether the client is allowed access. If the service ticket is correctly decrypted, the target server knows that the only other agent that could have issued it is the Security Service, since it is the only other agent that has access to the private key. All agents in the cell implicitly trust the Security Service, and many precautions are taken to ensure that this trust is not misplaced. The client uses its TGT to authenticate itself to the security service when requesting a service ticket, in a protocol exchange very similar to the service protocol just described. [Kohl 1989] describes the Kerberos protocol in more detail.

DCE does not provide a standard ACL manager; instead it requires each service to implement its own. This is a strength, in that it is very flexible, allowing service providers to define their own resources and control access to them individually; and it is a weakness, in that it leads to redundancy and makes developing a DCE server more difficult. To mitigate the development cost, DCE 1.1 provides an ACL library-implementing code that is common across ACL managers. This simplifies the programmer's task significantly, leaving only a small amount of custom coding and integration work.

Applications communicating via DCE RPC may choose an appropriate level of security. The lowest level is *authentication only*, followed by *integrity guarantee*—an encrypted checksum is provided with each message so that contents can be verified—and the most secure is *full privacy*, where the entire message contents are encrypted. Each level incurs a performance penalty.

DCE 1.1 supports the Generic Security Service (GSS) API. This is a standard programming interface to security services (described in Internet RFCs 1508 and 1509), intended to supplement existing Internet protocols. Using the GSS API, programmers can develop applications that use a subset of DCE security services through a standard, portable interface.

Overall, security is one of DCE's strongest points. The primary risk—credential stealing on multi-user machines—is rarely a problem in DCE's market niche. The RPC authentication and encryption options, built on mature Kerberos technology, are very attractive features for business software developers. DCE security is widely considered to be sufficient even for "business critical" software systems.

3.4.3 Time Synchronization

The third component of the core DCE services is the Distributed Time Service, DTS. DTS is provided to synchronize the clocks of all DCE hosts, which is required by a variety of protocols. DTS can synchronize to a *floating* time value—that is, one that is arbitrary but uniform—or, can derive its base "correct" time from an external source.

Including DTS as a core DCE service was an odd decision. Although time synchronization is a prerequisite for supporting the DCE core services, there is no direct dependence on any DTS API. Alternative methods for synchronizing clocks (e.g. NTP) are available and widely used both within DCE cells and elsewhere.

3.5 Application Development

DCE is solidly based on a client-server RPC programming model [Shirley 1992]. Server processes export procedure interfaces and service remote calls using DCE threads (based on POSIX 1003.1c, draft 4, threads, an early version of pthreads). Clients look up service names using CDS, which yields the binding information necessary for a client stub to make a connection to a server. DCE RPC supports several levels of security, from simple authentication of the client and server to complete encryption of the protocol exchange.

Like many RPC-based systems, DCE uses an Interface Definition Language (IDL) to specify the procedure interface. DCE IDL is similar to other interface languages, including CORBA IDL, but it is unique (there is no standard for IDLs). One use of DCE IDL is identifying an interface—that is, a network provided service. An interface has a textual name and associated identification attributes: for example, a unique identifier and major and minor release numbers. Identification attributes are used to determine whether clients are compatible with servers.

A second function of IDL is describing the data structures that will be transferred across the network. IDL provides a C-like syntax for describing data. (Types include all primitives; e.g., generic bytes, integers and floating-point values of several sizes, 8-bit character data, and pointers, as well as arrays, strings, structures, and unions.) All types are analogous to the C data structures with the same names. These type descriptions are used by the run-time RPC code to marshal and unmarshal RPC parameters automatically, hiding the details of network data formats from the application developer.

IDL is also used to define procedure interfaces. IDL procedure declarations specify the number, names, and types of arguments, as well as the "direction" in which each is transferred (a performance feature that prevents unnecessary data exchanges).

Typically, DCE application development is a three-part process. First, the programmer designs the service interface and writes an IDL description of the required data types and procedural interfaces. This is fed into an IDL compiler, which generates link-ready code to be used later. The programmer then writes the clients and server, in either order. Clients implement the main control flow and call RPC stubs. The server consists largely of procedure bodies, with some control code. DCE servers are typically multithreaded using DCE's threads package, and as each RPC is received a thread is dynamically dispatched to service it. The client stubs and server are each compiled and linked with the IDL compiler output, producing execution-ready client and server programs.

At run time, the server registers its interfaces with CDS. Clients then identify services by name and, assuming that the service attributes match appropriately (for example that the version numbers match within conventional requirements), they *bind* to the service interface. This allows the client to make RPCs, with the procedure body executed by server-side threads, without knowing the details of parameter marshaling or network communication. CDS's location-hiding and binding group features facilitate the development of fault-tolerant and load-balanced services.

Currently, only the C language is directly supported by DCE IDL. However there are several third-party packages that facilitate the development of object-oriented distributed systems using DCE. HP has a product called OODCE that The Open Group plans to incorporate into a subsequent DCE release. Chisholm Technologies provides a Java-class library for DCE called JDCE. Citibank freely distributes two C++ libraries for DCE: Objdce, which encapsulates core service features, and Objtran, which provides Encina-based transaction processing objects. (Encina is a transaction processing monitor built on DCE's core services, and is described in more detail below.) The Open Group also promises a future version of DCE that includes IDL support for C++.

3.6 DCE Applications

3.6.1 General

DCE supplies infrastructure, rather than applications. Hence its deployment is driven by the need for the applications it supports, or by the need for an environment upon which to build a custom distributed system.

There are relatively few DCE applications available. The majority are application development aids that try to reduce the complexity of the DCE interface or leverage DCE services for a more specific purpose. A few fill niche needs within established DCE environments, for example providing printing services. (The Open Group's web site includes a list of DCE and Encina applications.)

Two DCE applications of note, discussed in detail subsequently, are the Distributed File Service (DFS) and Encina, a transaction processing support package. And strictly speaking, neither of these are applications either; both provide additional higher level infrastructure used for developing custom distributed systems. Of the two, the Encina seems to drive DCE deployment much more than DFS. In the information systems market there is a strong

desire for secure distributed transaction processing infrastructure upon which to build *business-critical* software. Encina's relative success suggests that it satisfies these needs well.

The relative dearth of off-the-shelf DCE applications, and the fact that the most popular one is itself an infrastructure product, means that most DCE deployment is driven by the need to establish a base for custom distributed systems. As one would expect given the nature of Encina, these custom systems are almost all supporting business applications. (The Open Group's web site includes a list of end users and their plans for DCE.)

3.6.2 Legacy Code

One factor limiting the proliferation of DCE is its inability to support *legacy* TCP/IP applications. In general, to take advantage of the DCE infrastructure an application must be written specifically to use the DCE APIs. There are some exceptions—for example, an application that uses the GSS API for security can work without modification in a DCE cell. But this is little help, since there are relatively few extant applications supporting GSS API and because the GSS API is primarily concerned with secure messaging. Thus there is an extensive library of TCP/IP applications that cannot take advantage of the DCE infrastructure.

A relatively new product from IntelliSoft Corp., called DCE/Snare [Chinitz 1996], provides a way to support legacy TCP/IP services. It does this by intercepting network packets, and in some cases by setting up secure tunnels and redirecting the packets. Security managers can set up access control policies that limit connections based on location, identity, and time, and that are protocol/host specific. There is also a DCE/Snare-lite downloadable module that allows non-DCE clients (i.e. desktop machines) to authenticate to the DCE cell (by setting up a public key encryption tunnel). DCE/Snare is a significant step forward in making the DCE Security service generally useful—it largely alleviates the problem of requiring custom re-writes of existing tools.

3.6.3 DFS

The Distributed File Service (DFS) is a DCE application that supports distributed storage [OSF 1991, Kramer 1996]. DFS is based on Transarc Corporation's AFS product, which was developed jointly by IBM and Carnegie Mellon University (CMU). AFS has been used as the primary production file system at CMU, and following its early success it was *productized* by Transarc and is now a relatively successful commercial product. DFS is essentially AFS re-written to work within the DCE RPC framework. Transarc now sells DFS as well as AFS.

DFS presents a global, hierarchical file namespace to its clients. The name hierarchy is composed of sub-trees welded together at *mount points*. Each sub-tree physically resides on a unit of storage called a *fileset.* Filesets can be cloned, meaning that a copy-on-write duplicate can be created (useful for backups), or replicated, meaning that a complete, read-only copy is created. DFS supports on-disk client-side caching, guaranteeing cache coherency using a token exchange protocol.

DFS takes care to hide fileset location information from clients. To find a fileset (for example during traversal of a pathname, when a mount point is crossed), clients use the fileset name information stored in the mount point to look up the fileset's location in a database. The client then talks directly to the fileset server. This location hiding, in combination with fileset replication, provides the basis for supporting load balancing and fault

tolerance for DFS services. If one copy of a replicated fileset is not available, the Fileset Location Database simply points the client at another.

3.6.4 Encina

Encina, like DCE, is an infrastructure product [Houston 1996]. It is a layer of software that uses the core DCE services to provide a transaction-oriented business software development environment. Encina supports a three-tier client-server architecture, currently en vogue in the information systems business. The first tier is client nodes, running user interface software. The second is application servers, providing services to the client nodes and managing access to the third tier, data resources (usually databases running on mainframe systems).

The core of Encina is Encina Monitor, which supports reliable, secure transaction processing. Reliability and security are inherited from the underlying DCE services. Other components of Encina include the Peer-to-Peer Communications (PPC) package, which supports interoperability between client-server and mainframe systems (i.e. between the second and third tiers) and the Recoverable Queuing System (RQS), which supports fault tolerant batch processing of large or complicated transactions. The Structured File Server (SFS) provides an interface with record-based (ISAM/VSAM) data management systems. And Encina provides an object oriented development environment with Encina++.

3.7 DCE Experience and Commentary

The DCE Security service and CDS together provide the basic functionality required by all RPC-based distributed applications. They are individually strong components and passably well integrated. DTS is extraneous in many computing environments, but it is easily deactivated. DFS provides many attractive features as a distributed file-system, and it is reliable and performs well under typical UNIX file I/O loads. Encina is an attractive transaction processing monitor that takes good advantage of the underlying core DCE services.

But despite these positive features, DCE has not been as successful as some industry analysts expected. Our experience at the Numerical Aerodynamic Simulation (NAS) Systems division at NASA Ames Research Center points to a number of characteristics that have retarded its general deployment:

- Bloat. DCE is huge and complicated. This is not implicitly a fault, but the consequences are serious. DCE is difficult to learn, from either a programmer's or administrator's point of view. DCE reseller support personnel face similar obstacles in building expertise and in our experience are unlikely to be helpful, leaving customers to wade through the incomplete documentation and vague error messages on their own. Administrative problems are hard to isolate and correct, and the process is time consuming. The large amount of code led to a lot of early bugs. Again because of the size and complexity of DCE, isolating and fixing the bugs is an ordeal. In mature DCE implementations, for example IBM's, many of these bugs have been fixed over time. In newer implementations, for example SGI's, there are likely to be many remaining. (Because reference source releases from the Open Group are relatively rare, DCE resellers end up making many modifications to their own implementations to correct problems reported by their customers; this means that if vendor A fixes problems, the same problems will remain in vendor B's implementation. Furthermore, vendor A has little incentive to make its bug fixes available to other DCE resellers, because they are in competition. This

also leads to divergence of the DCE source code, which in turn makes bug fixes less transferable.)

- Cost. Not so much in terms of expenditure for the DCE software itself, but rather the personnel and training costs of deploying and maintaining a DCE cell. DCE is pervasive. In order for a site to take advantage of it, the core service client software must be installed on every machine. This means that a substantial collection of software has to be installed, configured, and then maintained. And if applications are to be developed on a client, the IDL compiler and other development tools also must be installed. Because of the size of DCE, as discussed in the previous item, there is a steep learning curve that presents a significant obstacle to users, developers, and administrators. This adds up to a substantial commitment in effort and training to deploy DCE.

- Lack of object support. As discussed above, DCE does little to support an object oriented approach to distributed computing. Strictly speaking it can be used to implement object oriented systems, just as any procedural programming language can be used to build object oriented software, but it was not designed to support the object model and does not do a good job of doing so.

- Poor diagnostic messages. Combined with the size of the code, this is particularly painful. DCE cell problems are typically discovered when one or more services simply stop working. If error messages were logged, which all too often is not the case, they are rarely useful. Typically, solving a DCE problem involves hunting fruitlessly for several hours before giving up and reconfiguring the software for the broken system.

- Poor documentation. Again, combined with the size of the code, the lack of clear, accurate documentation is a serious problem. For example, the API documentation is not complete—the semantics of some function parameters are not properly defined. This puts the application programmer in the frustrating situation of having to take a guess and try to verify it by experimentation (which—given the size and complexity of the APIs and the poor diagnostic messages—is a difficult task at best). Administrative documentation is slightly better, primarily because most resellers have integrated the procedures into their own administrative model and written their own documentation. But much room for improvement remains.

- Lack of a single programming model. DCE is an amalgamation of individual distributed computing infrastructure components, and it shows. The seams between services are clumsy and lead to an inconsistent, difficult API. Many programmers find the environment distasteful.

- DTS. DTS re-implements features that are already provided by NTP and other clock management software, which at many sites are already deployed. Despite this, the Open Group continues to treat DTS as a core service, causing problems in DCE installation and administration. Furthermore, it has historically been one of the less reliable DCE core services, with bugs that not only disable DTS but can also have a negative effect on other applications running on DCE clients.

- Incompatibility. DCE itself provides no way to support *legacy* TCP/IP applications. Programmers are expected to re-implement basic services if they are to take advantage of the DCE security service, for example. This is a major problem, although a relatively new package called *DCE/Snare* helps alleviate it.

Despite these unattractive features, some companies have made a major commitment to DCE. Reading through The Open Group's lists of DCE case studies, it quickly becomes clear

that the majority of DCE sites are planning to build their own business-oriented distributed systems on top of the base DCE services. The combination of security, support for load balancing and fault tolerance, and in many cases a desire for the Encina package together make DCE attractive middleware for business software development. These companies are making a serious commitment to DCE. The problems of a steep learning curve and difficulty supporting legacy applications are mitigated because of the scale of the proposed projects. Since a heavy investment would be required regardless of whether DCE were used, DCE's costs are less of an obstacle. And because this type of user is interested in building an integrated suite of applications based on DCE, legacy code issues are less important.

For scientific or academic computing, the arguments in support of DCE are much weaker. The startup costs are high, both from an organizational point of view and from the point of view of individual developers. A commercial information systems group supporting a suite of custom built DCE-based applications can afford to specialize on an esoteric and complicated infrastructure. Academic and scientific users are less likely to be willing to invest the time and effort required to learn to use DCE. Because there are virtually no extant applications, and the need for secure transaction oriented computing is much less pronounced, this class of user has little need for DCE. If legacy TCP/IP applications were easily supported then DCE might find a role as security infrastructure, but there is currently superior support for Kerberos, functionally equivalent to DCE's security component. Object based technology is strongly favored in today's distributed systems community, and unless one is building transaction oriented software using Encina++ DCE does not support object oriented programming well. For most academic or scientific sites, DCE simply does not offer enough value to outweigh its cost.

4. CORBA

4.1 CORBA Overview

The Common Object Request Broker Architecture (CORBA) is a standard for transparent communication between application objects [OMG 1995, Orfali 1996b, Umar 1997, Vinoski 1997, Yang 1996]. The CORBA specification is being developed by the Object Management Group (OMG). The OMG is an non-profit industry consortium of over 700 software vendors and users involved in the development of object technology for distributed computing systems. The OMG does not produce any software, only specifications which come from OMG members who respond to Requests for Proposals (RFP).

In this section, we first give a brief overview of the Object Management Architecture (OMA) and its components, with an emphasis on the services that have been specified. This is followed by a detailed description of a key component of the OMA, the Object Request Broker (ORB). This is referred to commercially as CORBA, and in a broader sense, CORBA is becoming a label of all ORB-related technologies. Then we briefly describe the Internet Inter-ORB Protocol (IIOP) and list the main ORBs and their features. Finally, we discuss some of CORBA applications.

4.2 The Object Management Architecture

In November 1990, the OMG introduced a reference architecture for object oriented applications called the Object Management Architecture (OMA). It was revised several times in September 1992, January 1995, and January 1997. The OMA is mainly composed of an Object Model and a Reference Model [OMG 1997a, Vinoski 1997]. The OMA object model defines how objects are specified in a distributed environment. It is a client/server model

where the servers are objects that provide services to clients. The clients obtain services by invoking operations on server objects.

In the OMA object model, a client sends a request to an object through a well-defined encapsulating interface. The request is an event, and it consists of an operation, an object reference of the target object, zero or more actual parameters, and an optional request context. The interface determines the operations that clients can perform using the object reference. The object reference is an object name that identifies the same object each time the reference is used in a request. In this model, the clients are isolated from the object implementation.

The OMA Reference model, based on the latest version [OMG 1997a], defines the OMA components, their interfaces, and the interactions between these components, see Figure 3. These components are:

- Object Request Broker (ORB)
- Object Services
- Common Facilities
- Domain Interfaces
- Application Interfaces

The ORB, a key piece of the OMA, enables clients and objects to communicate in a heterogeneous distributed environment. It provides transparency of object location, activation, and communication. We will discuss the ORB in more detail in the next section.

Object Services, which sit close to the ORB, provide basic services for using and implementing objects. These are general services that are used by many distributed object applications. They augment the functionality of the ORB so that the ORB can be lightweight and efficient while these services can be added as needed. Currently, the following services have been specified by the OMG [OMG 1997b]:

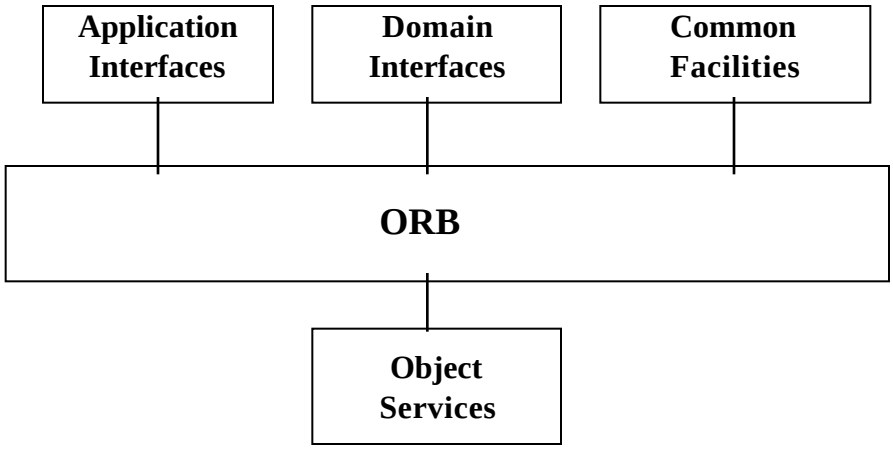

Figure 3. The OMA Reference Model.

1. **Naming Service**: It allows components to locate each other by name - by mapping (binding) names to object references.

2. **Event Service**: It allows components to dynamically register or unregister to an event. It supports both push and pull delivery models; i.e., consumers can either request or be notified of events.
3. **Life Cycle Service**: It defines operations for creating, deleting, copying, and moving objects.
4. **Persistent Object Service**: It allows the state of an object to be saved in a persistent store and restored when it is needed. It provides interfaces for storing objects in data stores such as object databases, relational databases and flat files.
5. **Transaction Service**: It provides the two-phase commit mechanism among components using either flat or nested transaction models.
6. **Concurrency Control Service**: It enables multiple clients to coordinate their access to shared resources by obtaining and releasing locks.
7. **Relationship Service**: It provides a mechanism for components to establish dynamic associations between each other.
8. **Externalization Service**: It provides a mechanism for getting data in and out of components in streams.
9. **Query Service**: It provides mechanism to invoke queries on a collection of objects using SQL-92 and Object Query Language (OQL-93).
10. **Licensing Service**: It provides operations for metering the use of components and charge accordingly, in order to control the intellectual property of their producers.
11. **Property Service**: It provides mechanism to dynamically associate named values with objects.
12. **Time Service**: It provides mechanism to synchronize time in a distributed object environment.
13. **Security Service**: It provides a framework to protect objects from unauthorized users. It comprises identification, authentication, authorization, access control, auditing, confidentiality protection, and non-repudiation.
14. **Object Trader Service**: It allows objects to publicize (export) their services while the clients lookup service offers.

Future services include: Archive, Backup/Restore, Change Management (Versioning), Internationalization, Logging, Recovery, Replication, Startup, and Data Interchange Services.

Common Facilities, which sit close to the user, are interfaces for horizontal end-user-oriented applications that can be used by many application domains. Four major common facility domains have been identified by the OMG: User Interface, Information Management, System Management, and Task Management.

Domain Interfaces (approved just recently by the OMG [OMG 1997a]) are oriented towards specific application domains such as finance, health, manufacturing, telecommunications, and education. Unlike the object services and common facilities, these interfaces are vertically-oriented.

Finally, Application Interfaces are interfaces specific to a certain application. These interfaces are non-standardized, and may be built from other objects such as Object Services and Common Facilities.

Another concept in the OMA Reference Model is Object Frameworks. These are a collection of objects that provide functionality to a specific domain and may contain any number (including zero) of Service objects, Facility objects, Domain objects, and Application objects.

4.3 The ORB

The CORBA 1.1 specification, introduced in December 1991, describes the interfaces and services that ORBs must have; i.e., CORBA is basically the technology adopted for ORBs. CORBA provides a clean model where the interface of an object and its underlying implementation are separated; clients do not need to know how or where servers are implemented. Server objects are visible only through interfaces and object references.

The CORBA 2.0 specification, introduced in December 1994, addressed the interoperability problem between different ORBs and extended some of the ORB features. As shown in Figure 4, this specification [OMG 1995] has the following components:

- ORB Core
- Interface Definition Language (IDL) Stub
- IDL Skeleton
- Dynamic Invocation Interface (DII)
- Dynamic Skeleton Invocation (DSI)
- Interface Repository
- Implementation Repository
- ORB Interface
- Object Adapters

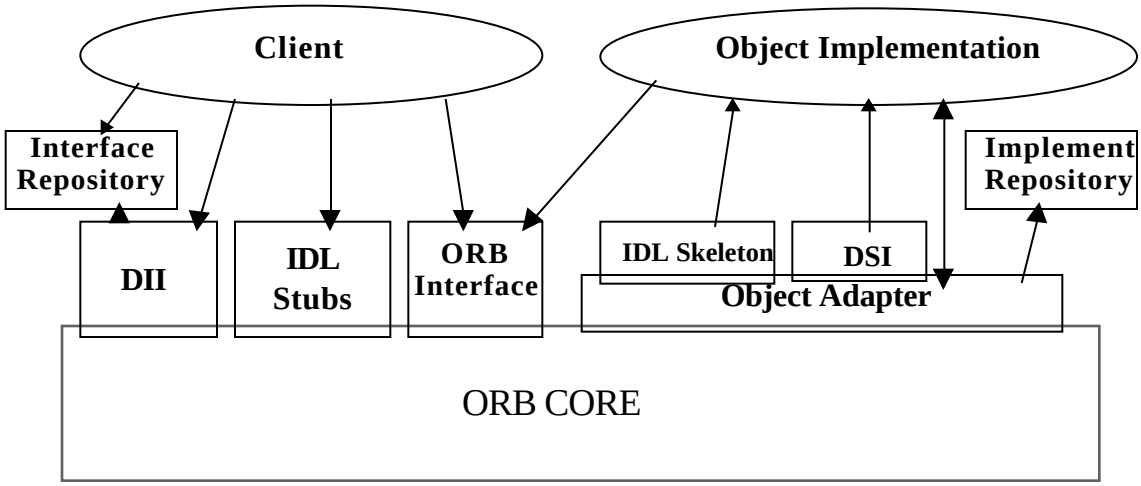

Figure 4. The ORB 2.0 architecture.

The ORB Core handles requests for remote objects. It uses object references to locate objects, activate them (if they are not already active), deliver the request to them, transfer control to them, and finally return any output values to the client.

IDL is a declarative language for describing the interfaces of CORBA objects with a syntax resembling that of C++. It is a subset of C++ with C++ implementation constructs removed and with extensions for distributed programming. It supports multiple interface inheritance. IDL interfaces contain attributes and operations used to define services provided by objects. An IDL compiler maps IDL constructs into a specific programming language (currently C,

C++, Ada, Smalltalk, COBOL, and Java) based on CORBA language bindings. In the IDL to C++ mapping, for example, each interface is mapped into a C++ class while each operation is mapped into a C++ member function. Each read-write attribute is mapped into two member functions (get and set) whereas each read-only attribute is mapped into a get function only.

The IDL compiler generates the client and server code, called client stubs and server skeletons, needed to implement the interface. The client IDL stubs are precompiled stubs that make calls to the ORB core on behalf of a client. They work with the client code to perform a request marshalling - converting the request from its programming language representation to a transmission representation. In essence, the stub acts like a local call or a proxy for the remote target object.

The server skeletons deliver requests to server objects. They unmarshal the requests and invoke specific server methods. The IDL stubs and skeletons are parts of the client application and server object. This means that they have a previous knowledge of the object interfaces. This method of invocation, through stubs and skeletons, is called static invocation.

Another invocation method is through the DII, and is called dynamic invocation. The DII is a mechanism for clients to make calls on objects with no compile-time knowledge of their interfaces. It allows applications to discover methods to be invoked at run time. It usually performs this function by consulting the Interface Repository. Dynamic invocation is usually more flexible but more complicated and less efficient than static invocation.

The DSI, introduced in CORBA 2.0, provides a run-time binding mechanism for servers. It has a similar functionality to the DII, at the server side, since it allows servers to handle requests for components that do not have skeletons. It can receive both static and dynamic invocations.

The Interface Repository provides a dynamic representation of available object interfaces in the distributed environment. It contains information about the registered object interfaces, their methods, and the parameters used for invocation. Applications can dynamically access, store, and modify this information.

The Implementation Repository contains information that allows the ORB to locate and activate server objects. It may also contain information related to the implementation of ORBs, such as debugging information, security, and administrative data.

The Object Adapters provide the means for object implementations to access ORB services. Among the ORB services are object reference generation, object operation invocation, activation and deactivation of objects, and security. CORBA specifies that a standard adapter called the Basic Object Adapter (BOA) should be provided by every ORB.

The ORB Interface goes directly to the ORB for operations that are common across all objects. There is a small number of these operations since most operations are through stubs, skeletons, and object adapters. This interface has a few APIs to local services that may be of interest to an application.

The CORBA components, which collectively are also called the ORB, do not reside in a single location. Portions of the ORB - such as the IDL stub, the DII, and the Interface Repository - are at the client side while the other portions - such as the IDL skeleton, the DSI, the Implementation Repository and the object Adapters - are at the server side.

Object invocation in CORBA can be done synchronously (blocking), asynchronously (non-blocking), or one-way (best-effort). In a synchronous communication, the sender sends a request and waits until the request either completes or fails. In an asynchronous communication (called deferred synchronous communication in CORBA terminology), the sender sends a request and proceeds with other work (does not wait) but it must check periodically for a response. Asynchronous communication is supported under the DII only. In a one-way communication, the sender sends a request and proceeds with other work without checking for a response. Here the receiver does not return a value to the sender.

4.4 IIOP

A key component of the CORBA 2.0 specification is the ORB Interoperability Architecture that provides interoperability between two different ORBs [OMG 1995]. The architecture has two main components: direct ORB-to-ORB interoperability and bridge-based interoperability. Direct interoperability between two ORBs is possible if the two ORBs share the same domain. Here a domain is where a distributed transparency is preserved. A bridge is used to map ORB information from one ORB to another.

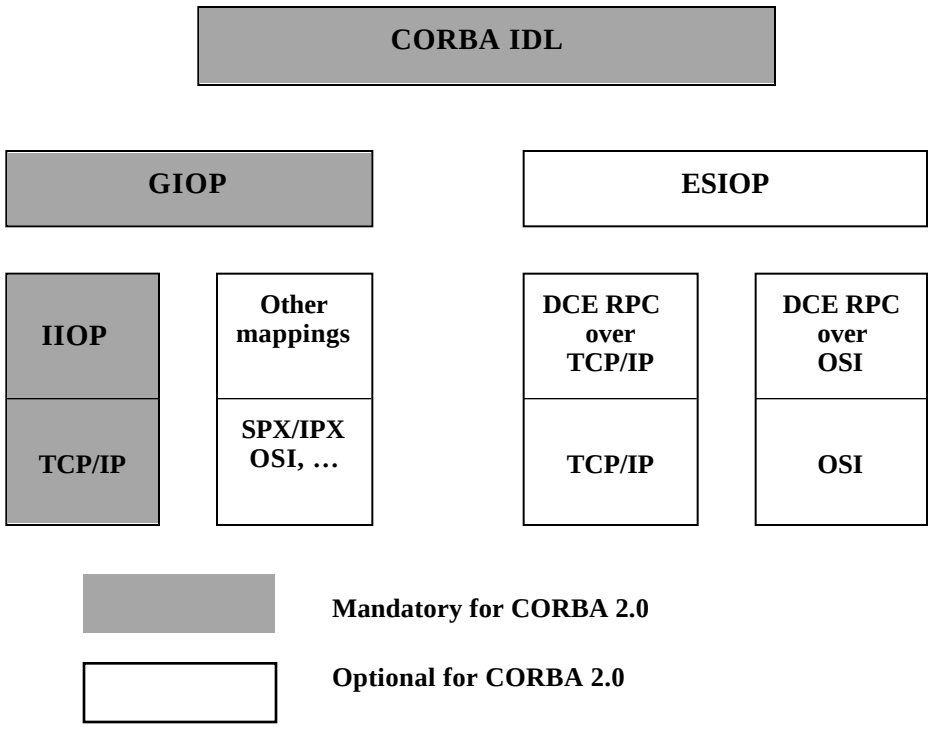

Figure 5. CORBA 2.0 Inter-ORB Architecture.

The interoperability architecture is based on the General Inter-ORB Protocol (GIOP), which specifies a set of message formats and a low level network data representation (called Common Data Representation) for communication between ORBs, see Figure 5. The GIOP mapping onto the TCP/IP transport layer is called the Internet Inter-ORB Protocol (IIOP). Support for both GIOP and IIOP is mandatory in CORBA 2.0 ORBs. Also, any CORBA 2.0 ORB must either implement IIOP natively or provides a *half bridge* to it. A half bridge is used to convert between a proprietary ORB and a standard CORBA 2.0 ORB (based on IIOP).

An alternative to GIOP is the Environment Specific Inter-ORB Protocols (ESIOPs), which is optional in CORBA 2.0. These protocols are specified for specific environments such as DCE which is called the DCE Common Inter-ORB Protocol (DCE-CIOP). The DCE-CIOP uses the GIOP Common Data Representation to represent CORBA IDL data types over the DCE RPC, which uses the native Network Data Representation.

Another feature of the interoperability specification is the standard Interoperable Object Reference (IOP) format. This standard format allows clients to locate objects created by any CORBA 2.0 ORB.

4.5 Commercial ORBs

There are at least a dozen ORBs from different vendors in the commercial market nowadays [OMG 1997c]. Although they differ in the number of platforms supported and the number of CORBA services provided, they are all based on the CORBA 2.0 specification and provide similar basic services. Also, most of them plan to provide the other services in the near future. The list of ORBs includes: Orbix from IONA Technologies; VisiBroker from Visigenic; ObjectBroker from BEA Systems, Inc.; PowerBroker CORBAplus from Expersoft Corp.; NEO from Sun Microsystems; Component Broker Connector from IBM; DAIS from ICL, Inc.; ORBplus from HP; Distributed Object Management Environment (DOME) from Object-Oriented Technology (OOT); and Inter-Language Unification (ILU) from Xerox Palo Alto Research Center (PARC). In this section, we discuss Orbix in more details, since we have some experience with it, and then we list the main features of some of these ORBs.

Orbix is a library based implementation of the CORBA specification from IONA Technologies Ltd [IONA 1996]. It is implemented mainly by two sets of libraries (client and server libraries) and a daemon, *orbixd*. The server library can send and receive remote object requests while the client library can only send requests. The daemon needs to run on the server's host side so it can start server processes dynamically. It also has an IDL compiler and supports interface and implementation repositories, static and dynamic method invocations, and many vendor-specific utilities. Language bindings are available for C++, Java, Smalltalk, and Ada95. The communication protocol for Orbix 2.x is IIOP.

An earlier study [Fatoohi 1997] compares the performance of Orbix 1.3 with the PVM message passing library [Geist et al. 1994] and BSD socket programming interface on a cluster of SGI workstations at NAS. The workstations are connected by four networks: a 10 Mbits/s Ethernet, a 100 Mbits/s FDDI, 800 Mbits/s HiPPI, and a 155 Mbits/s ATM. The study shows that the startup latency under Orbix is about three times that of the BSD sockets for the four networks (1.6 ms compared to 0.55 ms) and it is slightly higher than that of PVM. It also shows that Orbix achieves a reasonable throughput in traditional networks (6 Mbits/s on Ethernet and 77 Mbits/s on FDDI) while it suffers on high-speed networks 54 Mbits/s on ATM and 130 Mbits/s on HiPPI). Results for Orbix 2.0 for the same environment are comparable to the Orbix 1.3's results.

Table 1 lists some of the ORB features for seven ORBs. All these ORBs are CORBA 2.0 compliant and commercially available. They all provide Interface Repository and support both static and dynamic method invocations. Table 1 lists only the main platforms, the main language bindings and the main services. Other platforms (such as MacOS, Digital Unix, and Linux), language bindings (such as Smalltalk and Ada) and services (such as Life Cycle and Persistence) might be available for certain ORBs; see OMG [1997c] for a complete list. It is expected that many of the missing services will be available in the near future.

Table 1: The Commercial ORB Features.

Feature	IONA Orbix	Visigenic Visibroker	Expersoft CORBAplus	ICL DAIS	BEA ObjectBroker	Sun NEO	HP ORB+
Communication Protocol							
IIOP	·	·	·	·	·	·	·
DCE-CIOP					·		·
Language Bindings							
C				·	·	·	
C++	·	·	·	·	·	·	·
Java	·	·	·	·		·	·
Platforms							
Windows NT	·	·	·	·	·		·
Windows 95	·	·	·	·	·		·
Sun Solaris	·	·	·	·	·	·	·
HP UX	·	·	·	·	·		·
IBM AIX	·	·	·	·	·		
SGI IRIX	·	·		·	·		
OMG Services							
Naming	·	·	·	·	·	·	·
Event	·	·	·	·	·	·	·
Transaction	·		·	·			
Trading	·			·			·
Security				·			

4.6 CORBA Applications

CORBA has been used in many applications representing different industries. Among these industries are: aerospace and defense, banking and finance, manufacturing, health care, telecommunications, petroleum, transportation, insurance, and education. These applications range from small prototypes and experiments to mission-critical projects. Details of some of real applications are starting to appear in the open literature. In this section we discuss three interesting applications: Distributed Object Technology at Wells Fargo Bank [Ronayne and Townsend 1996], the Define and Control Airplane Configuration / Manufacturing Resource Management (DCAC/MRM) project at Boeing [Cleland et al. 1996] and the Digital Library project at Stanford [Paepcke et al. 1996].

Well Fargo has used the distributed object technology for interfacing disparate computer systems and providing a single consistent interface to customer information [Ronayne and Townsend 1996]. The problem that Wells Fargo, as well as many other banks, has is that different computer systems handle different bank accounts. In order to provide a

comprehensive profile of customer account information (by specifying the customer identity only), Wells Fargo used the ObjectBroker (from Digital and now owned by BEA systems) to integrate many legacy applications into a CORBA-based system. It uses the three-tier approach where the first tier belongs to the client (which provides consistent interface to the customer information), the middle tier belongs to the CORBA servers (which have the CORBA object implementation of business objects) and the third tier belongs to the back-end servers (which have mainframe applications). Wells Fargo started the project in late 1993, and placed it in full production use in less than four months. One key component of this project is the emphasis on the creation of re-usable components which can be combined in different ways to meet changing business requirements. Since its introduction, the project has been expanded to provide more services to the customers including delivering on-line banking services over the Web, providing on-line brokerage access, and adding Automated Teller Machines (ATMs) as client applications. The system is now in large scale production and processes more than 200,000 business transactions per day, which amounts to over one million CORBA method invocations per day.

The Boeing Commercial Airplane Group is using CORBA technologies to integrate Common-Off-The-Shelf (COTS) software and existing production systems in support of a major business process re-engineering initiative [Cleland et al. 1996]. The DCAC/MRM project seeks to redefine business processes to design, build, sell, and support commercial airplanes. Under the plan, each piece of data is owned, managed and updated in one place, called a Single Source of Product Data (SSPD). The DCAC/MRM project is using Orbix, by IONA, to integrate COTS software packages that will form the basis of new information systems at Boeing. These packages are integrated as a whole, by encapsulation, rather than breaking the applications down into smaller business objects. One goal of this project is a four-fold reduction in the product development cycle. Another goal is to enhance the company's capability to design new planes based on customer needs. The first phase of the project is supporting 5000 users at 15 sites. The second phase, which has just started, includes wider deployment and added components.

The Stanford Digital Library project is part of the National Digital Library Initiative: a four-year, $24 million initiative started in 1994 and sponsored by NSF, ARPA and NASA [Paepcke et al. 1996]. The initiative emphasizes the use of digital libraries not only for access and search but also for other services such as user communication about documents and copyright protection. The Stanford project addresses the problem of interoperability among heterogeneous services and repositories and uses CORBA to implement information access and payment protocols. The testbed will comprise five components: 1) client information interfaces, for handling user interaction with information in diverse formats; 2) information finding, for locating appropriate library services; 3) infrastructure and models, for developing models and supporting infrastructure for the interaction with documents and services; 4) economic perspective, for developing algorithms to detect partial overlap between reference documents; and 5) agents, for monitoring and transactions and retrieving information from the Web. The current testbed uses computing literature sources such as MIT press, the ACM, the World Wide Web, and many libraries. The implementation of the information-access protocol is based on ILU, an implementation of CORBA that is available free of charge from Xerox PARC.

5. DCOM

5.1 DCOM Overview

Distributed CCOM is an extension of of the Component Object Model (COM), developed by Microsoft Corp. as an object-based framework for developing and deploying software

components [Chappell 1997, Roy and Ewald 1997, Orfali et al. 1996b]. DCOM, previously called Network OLE, adds distributed support to COM to work across a network. The addition includes communication with remote objects, location transparency, and an interface to distributed security services.

COM is the core of many Microsoft's object-based technologies such as ActiveX and Object Linking and Embedding (OLE). ActiveX is now being promoted, by Microsoft, as a complete environment for components, distributed objects, and Web-related technologies. In essence, ActiveX is the Microsoft's label for a wide range of COM-based technologies. This set of technologies was formerly known as OLE but nowadays OLE refers to compound documents only.

Microsoft is currently giving control of some of ActiveX technologies to the Active Group, formed in association with the Open Group. This includes COM and DCOM but Microsoft has retained many ActiveX technologies, such as Web-related technologies.

COM-based technologies are directly supported by Microsoft on Windows NT, Windows 95 and MacOS. Other software vendors, such as Software AG and Digital, are currently porting some of these technologies to other platforms.

In this section, we first give an overview of COM then we describe DCOM, the middleware of ActiveX technologies, and some COM services, and finally we briefly describe two other components of ActiveX technologies: OLE Compound Documents and ActiveX Controls.

5.2 The Component Object Model

The Component Object Model (COM) is an object-based programming model that defines a common way to develop and access software components. COM, which has been part of Microsoft's Windows-based OSs for some time, separates object interfaces from their implementations, similar to CORBA. COM objects are designed to provide services through methods that are grouped into interfaces. This means that a COM object may support many interfaces, unlike a CORBA object which supports a single interface only.

Each COM object must support at least one interface, called *IUnknown*. All other interfaces inherit from this interface. The IUnknown interface has only three methods (*QueryInterface*, *AddRef*, and *Release* methods) to provide a way to get to other interfaces that an object supports and to control the life time of an object using reference counting. Reference counting is used by a COM object to count the number of clients using the object. When that number has fallen to zero, the object can be deleted.

COM supports only single interface inheritance, unlike CORBA which supports multiple interface inheritance. Also, COM does not support implementation inheritance. However, COM achieves object reuse through containment (delegation) and aggregation. In both cases, (containment and aggregation) the outer object encapsulates the interfaces of the inner objects and represents them to a client. The difference between the two is that in containment the outer object reissues the client's method invocations to the inner objects while in aggregation the outer object exposes the inner object's interfaces as its own.

Each COM interface is assigned a universally unique interface identifier (IID). Each COM object is an instance of a COM class, which has a unique identifier called a class identifier (CLSID). A client uses an IID and a CLSID to create an object and acquire an interface pointer. A client uses an interface pointer to invoke methods defined by that interface. Also, a client can acquire an interface pointer to a class factory, whose job is to create many

objects of the same class. Interface pointers are only valid when the object is running, since COM objects do not maintain state.

One or more COM classes are housed in a server, called a COM server. This server provides the necessary structure around an object to make it available to clients. There are three kinds of COM servers (see Figure 6):

- ***In-process servers*** execute in the same process as their clients. They are implemented as Dynamic Link Libraries (DLLs) under Microsoft's Windows-based OSs. The DLLs are loaded into the client process when a class within the server is first accessed by a client.
- ***Local servers*** execute in a separate process from their clients but on the same machine. Clients use a Lightweight Remote Procedure Call (LRPC) mechanism to communicate with local servers.
- ***Remote servers*** execute in a separate process on a different machine. Clients use DCOM to communicate with remote servers.

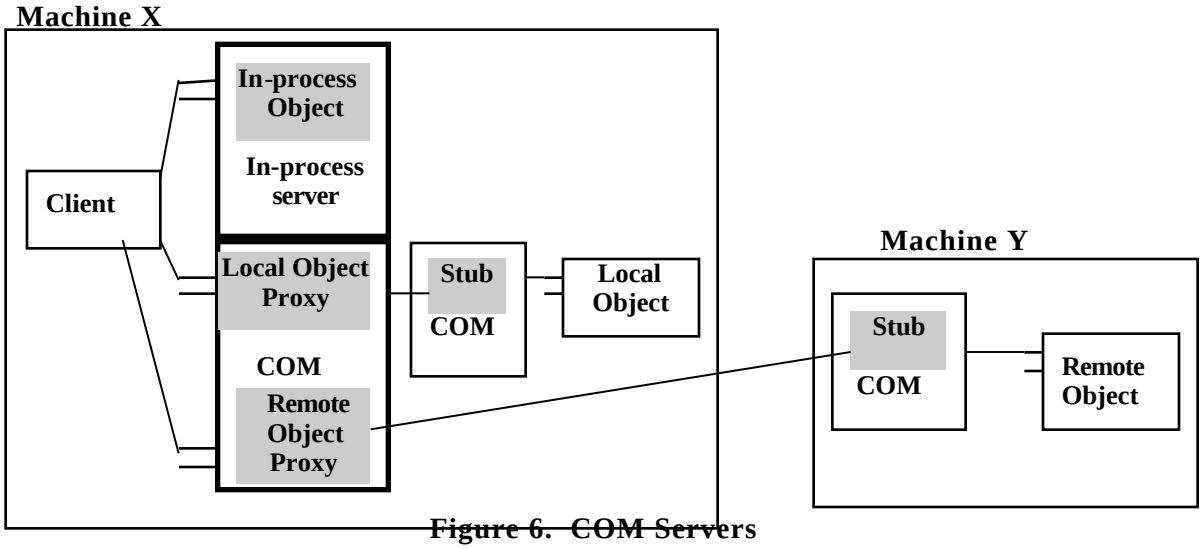

Figure 6. COM Servers

Local and remote servers, which are also called out-of-process servers, are packaged as separate executables (.EXE files). The client code for interacting with the three different servers is identical. This means that client applications need not be concerned with the server type (DLL or executable) or its location.

COM defines a binary call standard for interfaces. Clients use pointers to an array of function pointers, called virtual table (or *vtable*), to access an interface. For any given platform (hardware and OS combination), COM defines a standard way to lay out vtables in memory and a standard way to call functions through vtables. This allows vtable sharing among multiple instances of the same object class, which can reduce memory requirements. For in-process servers, vtable points to actual object implementation methods while for out-of-process servers vtable points to proxy methods. The binary standard allows the use of components without having access to the source code.

COM also defines a language for specifying interfaces called Microsoft Interface Definition Language (MIDL), an object-oriented extension of the DCE RPC IDL. This IDL, as with other IDLs, is used to define interfaces, the methods they support, and the parameters used by each method. An MIDL compiler is provided to generate client proxies and server stubs in

C or C++ from an interface definition, similar to client stubs and server skeletons in CORBA. The proxies contain the marshaling routines while the stubs contain the unmarshaling routines. Proxies and stubs are needed for out-of-process servers. For many standard interfaces, their proxies and stubs are usually provided (by Microsoft).

A COM interface is language-independent, like CORBA. However, in general, the language must support pointers to deal with vtables and interact with COM components.

In addition to providing specification for how objects and their clients interact through the binary standard of interfaces, COM is also an implementation contained in the COM Library. The COM Library includes a number of API functions to create COM applications. It also locates the implementation of a requested service and establishes a connection between the client and the server. COM offers this service by requiring that COM classes to register information about the server type and its location in a local registry. In addition the COM Library includes transparent remote procedure calls when an object is running locally or remotely. In Microsoft Windows, the COM Library is implemented as a DLL.

COM uses *Type Library* to store an object's type information (also called metadata), similar to Interface Repository in CORBA. Originally, another language called Object Description Language (ODL) was used to create type libraries. Recently, ODL has been superseded by MIDL which can now create type libraries.

Invoking methods in COM can be done either statically using vtables, similar to static invocation in CORBA, or dynamically using the *IDispatch* interface, similar to CORBA's DII. The use of IDispatch is also called automation, previously called OLE automation. Automation also allows COM objects to expose their methods to clients that do not support pointers, usually through interpreted languages.

5.3 Distributed COM

DCOM is an extension of COM that enables COM processes to run on different machines [Brown and Kindel 1996, Chappell 1997]. DCOM uses an extension of the DCE RPC for interactions between DCOM objects called Object RPC, or ORPC. The RPC extension includes a primitive data type to support object references and a parameter to each call for the remote object. The main difference between COM and DCOM is when the client asks the registry for the server location, the registry points to an IP address instead of pointing to a location on the local machine. ORPC can use either TCP or UDP but it favors the latter for scalability.

DCOM is integrated into Windows NT 4.0 and is available, as a free download, for Windows 95. Software AG has ported DCOM to Sun Solaris 2.5 and Digital Unix 4.0. Also, Software AG is currently BETA testing DCOM on IBM MVS and Linux 2.0 and begins testing on other platforms (such as HP-UX and IBM AIX) during the second half of 1997.

5.4 COM Object Services

COM and DCOM provide a number of facilities that are similar to CORBA services but they are not called services since they are built into the COM library. These facilities provide

functions similar to ones provided by CORBA services such as naming, security, transaction, persistence, life cycle, versioning, event, and data access services.

DCOM currently supports a scheme for identifying other systems by using a directory service such as the Domain Name System (DNS). DCOM will also support the Active Directory Service Interface (ADSI) which is basically combines DNS and the Lightweight Directory Access Protocol (LDAP). The ADSI is scheduled to ship with Windows NT 5.0 in 1998.

In addition, DCOM defines a standard interface to security services and provides multiple levels of security as needed [Chappell 1997]. It has several options for activation security and two main options for call-level security: automatic security and per-interface security. Activation security is achieved by supporting Access Control Lists (ACLs) on COM components. Remote servers can have an ACL that controls who is allowed to start a process. Also, ACLs can be managed with a tool. In automatic security, an object can set all of its security options, and all calls for this object will use these options. Per-interface security allows setting security options for different interfaces.

The Microsoft Transaction Server (MTS), previously code-named Viper, is a COM-based TP Monitor that uses DCOM for all object communication among machines. The MTS provides transparent transaction support to objects and manages threads, processes, database connections, and shared properties.

COM objects can provide several persistence models such as storing data in an ordinary file, in a *compound* file (also called *Structured Storage*), in a *property* bag (a bunch of properties), and in a location identified by a URL [Chappell 1997]. Clients can determine the persistence state of the COM objects for loading and storing.

The life cycle service is provided through the *IUnknown* interface. Also, the method *QueryInterface* provides some form of versioning service; see Orfali et al. [1996b] for details. COM's version of event service is called *Connectable Objects*. Connectable Objects, also called connection points, allow a two-way communication between an object and its client. These objects support outgoing interfaces, for calling back, as well as incoming ones.

Mcirosoft's OLE DB includes a set of component interfaces that provide single point of access for different types of databases. It uses Microsoft's Open Database Connectivity (ODBC) to access relational databases. Another Microsoft data access interface is ActiveX Data Objects (ADO) as language-independent, high-level programming objects built on top of OLE DB. While the OLE DB interface is exposed by data providers, the ADO interface is used by data consumers. OLE DB, ADO and ODBC can all work together in accessing data.

5.5 OLE

Microsoft's Object Linking and Embedding (OLE) is a technology that enables an application to create compound documents that contain information from different sources [Chappell 1997, Orfali et al. 1996b, Umar 1997]. Here a compound document is a container for data and components that come from different sources such as text editors, spreadsheets, graphics, and others. Another component document technology is OpenDoc which was originated by Apple and adopted by OMG, see Orfali et al. [1996b] for details.

OLE was first introduced in 1990 and was based on a protocol built on top of Dynamic Data Exchange (DDE). OLE 2 was introduce in 1993 and is based on COM.

OLE defines the interfaces between a container application and the server applications that the container manages. The container provides places for the servers to reside in. An application can be written as either a container only, a server only, or both. As the name implies, OLE provides two ways to access objects from other applications:

- Linking. A reference or a link to the object is placed in the container application while the actual object is not copied. The real object is accessed at run time so that changes made to it to be reflected in all container applications that have a link (an object linked simultaneously to multiple applications is allowed).
- Embedding. A copy of the real object is copied into the container application. Re-embedding is required to access a new copy of the object.

5.6 ActiveX Controls

Microsoft's ActiveX Controls, originally called OLE controls or OCXs, are COM objects that interact with a container in a standard way [Chappell 1997]. They are COM components implemented as DLLs, and basically similar to Java applets, described in the next section. Here a component is defined as a self-contained, software module (or an object) that performs a limited set of tasks within an application, has a well-defined interface for interoperability, and is distributed in an executable form. It is easier to use components than class libraries or frameworks since components usually support cross-language reuse and do not require recompilation or relinking to accommodate small changes.

ActiveX controls run inside control containers such as Web browsers, Visual Basic, Visual C++, and PowerBuilder. They can be written in many languages such as C, C++ and Java, and are binaries which may either already be present on the machine or may be downloaded from a Web server on demand. Unlike Java applets, controls are not *sandboxed*, which means that they are not insulated from direct contact with the host system.

There are already many ActiveX controls existing in the marketplace. They are commonly used on clients to implement spreadsheets, data viewing, and other applications.

6. Java

6.1 Java Overview

Java, developed by Sun Microsystems, is a label for a broad range of object-based technologies for the Web and distributed applications. It is an object-oriented programming language developed mainly for consumer electronics and Web applications. It is now becoming a computing platform for developing and running distributed applications [Kramer 1996]. This platform, called the Java Platform, consists mainly of the Java Virtual Machine (JVM) and Java API which includes Java core classes. The Java platform sits on top of many other platforms (hardware and OS) and compiles to *bytecodes*, which are machine-independent instructions for the JVM. The JVM is an abstract specification for a computing device which can be implemented in software or hardware.

Java programs can be either applets, applications, or JavaBeans. Java applets are small, or modular, programs that require a Java-compatible Web browser to run. An applet is downloaded and executed into the client's machine when the browser reaches a pointer to it. Java applications are stand-alone programs that do not require a browser to run. A Java application, like an applet, requires the Java Platform to run where the Java Platform can be a separate program or embedded within the OS.

JavaBeans are reusable software components that are platform-independent and allow dynamic interactions with other components [JavaSoft 1996]. Unlike a Java applet, which can only interact with its server and needs a browser to run, a JavaBean can interact with other components over the network and can run in containers other than a browser. While Beans are intended to be used mainly with builder tools, they can be used for other applications as well.

A Java program compiles into bytecodes that can run, using a Java interpreter, wherever the Java Platform is present. Java also supports regular compilers as well as *just-in-time* compilers that convert bytecodes into native machine language to speedup execution.

The Java Platform is currently embedded in browsers, such as Netscape Navigator, and run on many other platforms such as MS Windows 95, Windows NT, MacOS, Sun Solaris, HP UX, SGI IRIX, IBM AIX, IBM MVS, and Novell NetWare. In addition, many RISC chips, called the JavaChip family, will have the Java Platform and execute bytecodes natively [Kramer 1996].

The Java Language is the entrance to the Java Platform. It is similar to C++ but it was designed as a portable language for distributed applications. It is shipped as a product called the Java Development Kit (JDK) which contains: JVM, Java Class Libraries, and many Java tools such as Java Compiler, Java Interpreter, Java Applet Viewer, and Java Debugger. The current version of JDK, JDK 1.1, added new features such as JavaBeans APIs, Remote Method Invocation (RMI), and many Java services. JDK 1.1 is available on MS Windows 95 and NT, Sun Solaris, and many other platforms.

In the following sections, we give more details about the Java Platform, Java Object Model, JavaBeans, Java RMI, and Java Services.

6.2 Java Platform

The Java Platform has two main parts: the JVM and Java API [Kramer 1996, Linthicum 1997]. The Java API is a multi-platform standard interface to applets and applications. It specifies a set of interfaces for a wide range of applications. It is open so that third-party vendors can propose new APIs and work with JavaSoft in adopting them. Implementations of the API specifications are developed by JavaSoft as well as other vendors.

There are two types of Java APIs: the Java Base (Core) API and the Java Standard Extension API, see Figure 7. The Java Base API defines the core features for developing and deploying Java applets or applications. This API provides the basic languages, utilities, network, I/O, GUI, and applet services. The Java Standard Extension API extends the capabilities of the Base API. Some of these extensions may eventually migrate to the Base API. The Java Base classes and Java Standard Extension Classes, which are parts of the Java Platform, are the implementations of their corresponding API.

<table>
<tr><th colspan="2">Applets and Applications</th></tr>
<tr><th>Java Base API</th><th>Java Standard Extension API</th></tr>
<tr><td>Java Base Classes</td><td>Java Standard Extension Classes</td></tr>
</table>

JVM

Porting Interface

Figure 7. Java Platform.

The Java API is organized into groups or sets. Each set is implemented as one or more packages, and each package represents a group of classes and interfaces of related fields. Each set belongs to either the Base, Standard Extension, or a mixture (Base/Standard Extension). Some of the new Base API and Standard Extensions are described here:

- JavaBeans API. It is a Base API (in JDK 1.1), described below.
- Enterprise API. It provides connectivity to enterprise information resources such as databases and legacy applications. It currently encompasses four areas:
    - Java DataBase Connectivity (JDBC). It is a Base API (in JDK 1.1), described below.
    - Java RMI. It is a Base API, described below.
    - Java IDL. It provides interoperability and connectivity with CORBA. It includes a Java IDL, Language Mapping Specification, an IDL-to-Java compiler and a Java ORB core that supports IIOP. It is a Base API (out of JDK 1.1).
    - Java Naming and Directory Interface (JNDI). It is a Standard Extension API, described below.
- Java Security API. It is a Base API, described below.
- Java Media API. It supports interactive multimedia applications. It encompasses the following areas: Java Media Framework (including Java Media Player, Java Media Capture, and Java Media Conference APIs), Java Collaboration, Java Telephony, Java Speech, Java Animation, and Java 3D APIs. Most of them are Standard Extension APIs.
- Java Foundation Classes (JFC) API. It is used to build graphical User Interfaces (GUIs) for Java programs. In addition to the existing Abstract Window Toolkit (AWT), it includes Java 2D API and Swing Set API which are Base API.
- Java Server API. It is used to create server-based applications such as server administrations and dynamic server resource handling. It encompasses Java Servlet API and Java Server API which are Standard Extension API.
- Java Management API. It provides the building blocks for managing an enterprise network. It is an Standard Extension API.
- Java Commerce API. It is used for secure purchasing and financial management on the Web. An initial component is the JavaWallet, which defines and implements a client-side framework for credit card, debit card, and electronic transactions. It is a mixture of Base and Standard Extension APIs.

6.3 Java Object Model

Java does not define a language-independent object model, unlike CORBA and COM. It supports multiple interface inheritance, similar to CORBA [Orfali and Harkey 1997]. However, it supports single implementation inheritance, unlike CORBA and COM which do

not support implementation inheritance. Java separates object interfaces from their implementations, similar to CORBA and COM. Java objects, like COM objects, may support multiple interfaces.

Java provides automatic garbage collection which helps in memory management [Gosling and McGilton 1996]. Its memory management model is based on objects and references to them. Java does not support pointers. However, references to an object are through symbolic *handles* that are resolved to real memory addresses at run time. The memory manager keeps track of references to objects, and the object becomes a candidate for garbage collection when it has no more references. Also, Java supports multithreading by providing synchronization primitives. Java's multithreading capability makes it easier to provide portable, thread-safe code.

6.4 JavaBeans

JavaBeans is a software component architecture to create, assembly and use dynamic software components. The JavaBeans API defines a portable, platform-neutral set of APIs for components [JavaSoft 1997a]. These components could be used, as pre-built parts, in composing applications. They can also be plugged into other, non-Java, component architectures such as COM and OpenDoc. In addition, JavaBeans can be more like regular applications, which may then be composed into compound documents.

JavaBeans components can range in size and functionality from small GUI elements to sophisticated applications such as database viewers. These components are portable across all Java enabled platforms and can be used in many development tools and containers.

The main features of a JavaBeans component are properties, methods, and events. Properties are the attributes of a component that can directly affect or reflect the current state of that component. The methods are the normal Java methods that can be called from other components. Events provide a mechanism for components to notify other components about an event. In addition, JavaBeans provides many other services such as Publishing and Discovery, Persistence, and Component packaging, described below.

JavaBeans applications can interact with other applications across a network using several mechanisms including Java RMI (with a Java server for example), Java IDL (with a CORBA server for example), and JDBC (with a database server). Components can also migrate around the network in a sense that they can be created on one site and get forwarded around the network to other sites.

6.5 Remote Method Invocation

Java Remote Method Invocation (RMI) is a communication mechanism between distributed Java objects [JavaSoft 1997b]. RMI enables objects in one JVM to invoke methods on objects in a remote JVM. The method invocation on a remote object has the same syntax as a method invocation on a local object. A Java object can invoke a method on a remote object once it obtains a reference to that object. Such a reference can be obtained either from an RMI naming service or by receiving the reference as an argument or a return value. A client can call a remote object in a server, and that server can also be a client of other remote objects. RMI uses Object Serialization for marshaling primitives and objects.

The RMI system consists of three layers (as shown in Figure 8): the stub/skeleton layer, the remote reference layer, and the transport layer [JavaSoft 1997b]. The application layer sits on top of the RMI system. Each layer is independent of the next layer since the boundary between the layers is defined by an interface and protocol. Therefore, the implementation of each layer can be replaced without affecting the other layers.

The stub/skeleton layer is the interface between the application layer and the remote reference layer. This layer transmits data to the lower layer using a mechanism called Object Serialization which enables transparent transmission of Java objects between address spaces. Object serialization supports the encoding of objects into a stream of bytes. Non-remote arguments are passed by copy since object references are useful only within a single JVM. Remote objects, however, are passed by reference.

An RMI compiler (RMIC) generates stubs and skeletons, as in other middleware products. A client uses a stub (or proxy) for a remote object to invoke a method on that object. At the client-side, a reference to the remote object is a reference to a local stub. Such a stub implements all the interfaces that are supported by the remote object implementation. Therefore, a stub is responsible for initiating a call, marshaling arguments and unmarshaling the return value (if any). At the server-side, a skeleton is an entity that is responsible for unmarshaling arguments, making the call to the actual remote object implementation, and marshaling the return value.

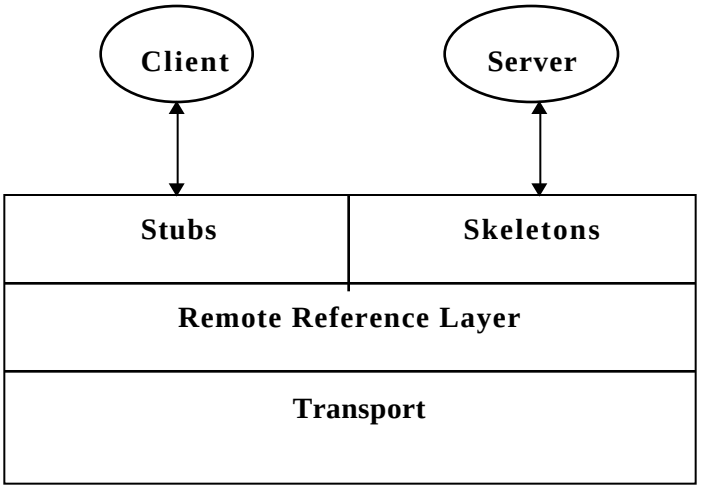

Figure 8. The RMI Architecture.

The remote reference layer is responsible for carrying out the semantics of the invocation, independently from the stubs and the skeletons. The RMI system supports different invocation mechanisms such as simple invocation to a single object (point-to-point) as well as invocation to an object replicated at multiple locations. In addition, it supports persistent references as well as non-persistent references. Moreover, it supports reconnection strategies, if a remote object becomes inaccessible. These different support mechanisms are carried out in this layer. As with the stub/skeleton layer, this layer has two cooperating components: at the client and the server sides.

The transport layer is responsible for setting up, managing, and monitoring connections. The current transport implementation is TCP-based but a transport based on UDP could be substituted.

In addition to Object Serialization, RMI uses another technique called dynamic stub loading to support client stubs that implement the same set of remote interfaces as the remote object. This technique allows the client to use the Java Platform's built-in operators if the client does not have a stub of the exact type.

RMI uses an algorithm called reference counting garbage collection to automatically delete remote objects that are not referenced by any client. Based on this algorithm, the RMI runtime keeps track of all references within each JVM. A *referenced* message is sent to the server for the object when the object is first referenced. When the last reference has been discarded, an *unreferenced* message is sent to the server. Remote objects are collected only when no more local or remote references exist.

Recently, Sun announced that Java RMI would support IIOP as a transport protocol in addition to its native transport protocol. It is adding IIOP interoperability to Java RMI which would be retaining its own features. Access to CORBA-based objects could be either through Java IDL or Java RMI.

6.6 Java Services

Java provides many facilities that are similar to CORBA services, but unlike CORBA services, most of these facilities are not well-defined services and are associated with different Java technologies. Some of them have been developed along with other Java technologies such as JavaBeans, Java RMI, and Java Platform. Here we briefly describe some of the well-known Java services. Among these services are security, JDBC, JNDI, Reflection and Introspection, Event, Persistence, and Java Transaction Service.

The Java Security API is a framework to create secure Java applets and applications. It provides security functionality such as cryptography with digital signatures, encryption, and authentication. It also provides support for Key management for secure database transactions and certificate facilities. It includes an abstract layer that applications can call. That layer calls the Java Security packages that implement the security feature. The framework also allows upgrading or replacing modules as needed. The current version of JDK, 1.1, contains only a subset of the security functionality, including APIs for digital signatures and message digests.

Java DataBase Connectivity (JDBC), a standard SQL database access interface, provides a uniform access to many relational databases from Java applications. JDBC is a set of Java classes that represents database connections, SQL statements, metadata, etc. JDBC, similar to Microsoft's ODBC, uses a driver manager to automatically load the right JDBC driver to access a given database. JDBC supports databases that produce a JDBC driver or a JDBC-ODBC bridge driver. A JDBC driver manager along with a bridge driver are included in JDK 1.1 while JDBC drivers are available for many databases such as Oracle, Sybase, Informix, and IBM DB2.

Java Naming and Directory Interface (JNDI) provides Java applications a unified interface to multiple naming and directory services, such as DNS, NIS, NDS, and LDAP. It also provides methods for performing standard directory operations, such as searching for objects using their attributes and associating attributes with objects.

The Reflection API enables a Java program to get information about the data fields, constructors, and methods of loaded classes, and to manipulate those fields or invoke those methods. This API, which is a Java Base API, is an essential part of JavaBeans. Introspection

is a mechanism through which the application builder tool analyzes a JavaBeans component and figure out its properties, events, and methods. It provides two mechanisms to accomplish this inspection at run-time: either using the low-level reflection mechanism or using vendor provided explicit information about the bean. The functionality provided by reflection and introspection is similar to that of CORBA Interface Repository and COM Type libraries.
Persistence, developed with JavaBeans, is a mechanism for saving the state of beans so it can be resurrected later [JavaSoft 1997a]. Normally, a bean will store the state of its exposed properties and any other beans that it may contain. However, a bean will not store pointers to external beans or to event listeners. Two persistence mechanisms are supported: either using Java Object Serialization, which provides an automatic way of storing the internal state of a collection of objects, or using *Externalization*, which allows a class full control over the writing of its state.

Events are one of the core features of JavaBeans [JavaSoft 1997a]. Events provide a mechanism for allowing beans to fire as well as capture events. This includes notifying beans if a desired event occurs. Java currently has two event models [Rodriguez 1997]. The first model is based on inheritance and propagation and available only in the AWT. The second model, called the Delegation Model, is more appropriate for JavaBeans. In this mode, events are propagated from a *Source* object to one or more *Listener* objects. When an event occurs, the source object generates an event object and sends it to the registered listener objects. Here only the registered objects will receive that event.

The Java Transaction Service (JTS) is mainly a Java mapping of OMG's Transaction Service (OTS) through the Java IDL interface compiler. Its initial version, 0.5, was released in December 1996.

7. Comparisons and observations

The four middleware technologies (DCE, CORBA, DCOM, and Java) provide somewhat similar functionality; however, there are some key differences between them. In this section, we compare these technologies based on a set of key features; see Table 2 for a summary.

Both DCE and CORBA, as middleware for distributed applications, have been around for several years. DCE in general is more mature and has a more complete set of core services than other middleware technologies. On the other hand, CORBA has a richer set of services than the others. However, many CORBA services have been approved just recently, and would take some time (about a year) to implement them. Also, many CORBA facilities and some CORBA services are still being debated by OMG and have not yet been approved. Even though the first CORBA specification was released in December 1991, the interoperability issue was not addressed until three years later, with the release of the CORBA 2.0 specification. It is expected that it will take about two more years (till the end of 1999) for CORBA to have a complete, mature and fully implemented set of services and some key facilities. But that should not prevent software developers to use CORBA now since it is the only general purpose middleware based on the object-oriented technology.

DCE suffers from a lack of object support in its architecture. It has no support for inheritance of any kind. Also, it is big and complex which make maintenance a non-trivial task. However, it offers the best security service among the four middleware technologies. Also, its other services and components, such as the CDS naming service and RPC, are quite good. It still remains to be seen whether DCE will survive as a middleware since the market is becoming more object-oriented.

Both DCOM and Java are owned, specified, and mainly developed by vendors, Microsoft and Sun respectively, with some input from others. The advantages and disadvantages of vendor-specified products, compared to standard committee specifications such as CORBA and DCE, are well known and understood in the industry. To mention a few here, multi-vendor standard committees tend to be slow in developing standards and some of these standards are very broad and based on many compromises, rather than adopting the best technical proposals. On the other hand, committee standards tend to be more inter-operable, vendor-neutral, and open to public comments and suggestions. These are clearly important issues in comparing middleware technologies from the two groups, such as CORBA and DCOM.

Table 2: Middleware Comparison

Feature	DCE	OMA/CORBA	COM/DCOM	Java/RMI
General				
Inception	1992	1991	1996	1996
Ownership & development	Open Group	OMG	Microsoft (& Active Group)	Sun
Standardization	open	open	vendor-control	vendor-control
Distribution form	implement	specification	implem/specific	specific/implem
Platform support	many	many	few (Windows)	many
Programming				
Model	procedural	O-O	O-O	O-O
IDL	Yes	Yes	Yes	No
IDL mapping	C	many	C, C++, Java	Java
Service I.D.	UUID	-	UUID	-
Communication Protocol				
Comm. protocol	RPC	IIOP	Object RPC	RMI/IIOP
Transport protocol	TCP	TCP	UDP	TCP
Object-Specific				
Interface per object	-	single	multiple	multiple
Interface inheritance	-	multiple	single	multiple
Implement inheritance	-	none	none	single
Object repository Info	-	Interface repository	Type libraries	Reflection/ introspection
Static Invocation	RPC	SII	vtable	built-in
Dynamic Invocation	-	DII	IDispatch	
Component SW	-	-	ActiveX control	JavaBeans
Services				
Security	Kerberos	Specification	incomplete	incomplete
Naming	CDS	Naming service	DNS & ADSI	JNDI
Transaction	Encina	OTS	MTS	JTS
Database access	Encina	OQS	OLE DB/ADO	JDBC

Another key difference between these middleware technologies is whether they are based on specification and/or implementation. CORBA is a pure specification-based middleware where OMG develops the specification and vendors implement it and produce products based on it.

DCE, on the other hand, is an implementation where vendors obtain the DCE code from the Open Group and customize it for their own system. DCOM and Java are more of a hybrid of the two approaches. Both Microsoft and Sun develop the specification and provide an implementation on their favorite system: Microsoft OSs and Sun Solaris, respectively. Other implementations are usually performed by third-party vendors. In the case of DCOM, if there are differences between its Windows implementation version and the specification, the Microsoft implementation is to be considered definitive, and the specification will be adjusted to match.

Among the four middleware technologies, only DCOM is not widely supported on multiple platforms. It runs mainly on Microsoft Windows OSs and a few Unix systems through third-party vendors. DCE, CORBA and Java enjoy wide support on several platforms including Microsoft Windows OSs.

The basic communication services for the four technologies, as defined in section 2, are very similar. They are all based on the remote procedure call paradigm, except that DCE RPC is procedural oriented while the others are object oriented. Also, the core services of the four middleware technologies are very similar, except that some of them are more mature than others. In addition, all of them use an IDL (even though a different IDL) to develop a distributed application. Moreover, the process of building information infrastructure using any of them is not that much different, since these middleware technologies are designed to achieve the same goal as outlined in section 2.

The history of a product plays an important rule in defining, shaping, and maturity of the product. Middleware is no exception. Although COM and OLE have been around for some time, they have gone through many changes in definition, specification and even in naming. For example, COM was not designed for distributed applications; it is an object model for a single machine and DCOM is simply *COM with a longer wire.* Also, Java started as an object-oriented language mainly for consumer electronics and the Web, and is evolving as a middleware for distributed applications. Both DCOM and RMI are add-on ORBs to an existing environment to provide support for distributed applications. They also provide a set of support services which are in early stages of development and may take a while to mature. CORBA, on the other hand, had a distributed object model (OMA) from the beginning, and was designed for distributed applications.

A combination of two or more middleware technologies is also a possibility. A good example here is a combined CORBA and Java middleware where Java provides the code foundation and CORBA provides the distributed object infrastructure [Orfali and Harkey 1997]. Here Java provides the language as well as the component architecture, JavaBeans, that CORBA lacks. CORBA, on the other hand, provides the communication protocol, IIOP, many well-specified services, and a clean object model. Java objects can interact with other objects, written in other languages, using CORBA/IIOP. There are even products, such as Visigenic's Caffeine, that can define CORBA objects using ordinary Java interfaces instead of IDL, thus eliminating the need for an IDL. One immediate application of such a middleware is the Object Web, the marriage of the distributed objects and the Web [Orfali and Harkey 1997].

Middleware services do not have to be provided as single components. Actually, there is little value in individual components since some core services are always needed. Many vendors are starting to bundle basic services - such as naming, messaging, data access, and security in addition to an ORB - and offer them as an integrated service. Such a service is similar to DCE. Middleware integration, of independent components into a system, is not a trivial task and requires a good knowledge of system engineering. Some components might be hard to integrate, since they were designed for a specific purpose, and some reengineering might be

needed. Other issues include portability, performance, complexity, reliability, and scalability of the whole system.

8. Conclusions

Middleware is a major enabling technology for the applications that require flexibility, portability, interoperability, scalability, and transparent distribution. It will continue to become more sophisticated since it allows users to access remote services as if they are local (transparently). Vendors and users are increasingly dependent on standards or de facto standards since they want to make sure that their investment in developing and integrating middleware services will pay off. Many standard committees, such as the OMG and the Open Group, are working on introducing standards into industrial products. Also, many major vendors, such as Microsoft and Sun, are pushing to make their products a de facto standard.

The trends are toward simplifying middleware and expanding its functionality into new applications. Simplification can come in the form of developing tools in order to ease the process of integrating and using different middleware services. These tools can be used to distribute objects and manage them in a large distributed system. Adding new services, that are common in many application domains, would attract new applications which will benefit from these services.

Middleware is a tricky business because there is a strong need for it but no dominant solution. Since middleware is an infrastructure, it is non-trivial to replace if one chooses incorrectly at the beginning. The middleware market may take some time to settle. In the mean time, middleware should be chosen based on the organization's needs. Current and future applications should be well defined and analyzed due to the complexity of middleware.

In this paper, we described four main middleware technologies and studied their features and the services they provide. Many other middleware technologies were not covered in this study. Among them are database-specific middleware and Message-Oriented Middleware (MOM); for information, see Orfali et al. [1996a] and Umar [1997]. Also, this is only an overview of these technologies. More work needs to be done in analyzing them in order to provide a comprehensive view of their functionality and their limitations.

References

Bellovin, S., et. al. 1991. *Limitations of the Kerberos Authentication System*, 1991 Winter USENIX Proceedings.

Bernstein, P. 1996. *Middleware: A Model for Distributed Systems Services*, Communications of ACM, Vol. 39, No. 2, pp. 86 -- 98.

Brown, N., and Kindel, C. 1996. *Distributed Component Object Model Protocol -- DCOM/1.0*, Network Working Group, Internet-Draft (Work in progress). Also in URL: http://www.microsoft.com/workshop/prog/com.

Chappell, D. 1997. *Understanding ActiveX and OLE*, Tutorial Notes, Object World West, San Francisco, CA.

Chinitz, J., et. al. 1996. *DCE/Snare; A Transparent Security Framework for TCP/IP and Legacy Applications*, 1996. Also in URL: http://www.isoft.com/snare.

Cleland, V., et al. 1996. *Boeing Commercial Airplane Group's Application Integration Strategy*, First Class (OMG magazine - retired). Also in URL: http://www.iona.com/Customers/boeing2.html

Fatoohi, R. 1997. *Performance Evaluation of Communication Software Systems for Distributed Computing*, Distributed Systems Engineering Journal, Vol. 4, No. 3.

Geist, A., et al. 1994. *PVM 3 User's Guide and Reference Manual*, Report ORNL/TM-12187, Oak Ridge National Lab., Oak Ridge, TN. Also in URL: http://www.epm.ornl.gov/pvm

Gosling, J. and McGilton, H. 1996. *The Java Language Environment: A White Paper*, JavaSoft, Also in URL: http://java.sun.com/docs/white/langenv

Hewlett Packard 1995, *HP OODCE/9000 Product Brief*, 1995.

Houston, P. 1996. *Introduction to DCE and Encina*, 1996. White paper prepared for Transarc Corp.

Hu, W. 1995. *DCE Security Programming*, O'Reilly & Associates, Inc.

IONA Technologies 1996. *The Orbix Architecture*. Also in URL: http://www.iona.com/Products/Orbix/Architecture/index.html

JavaSoft 1996. *Java Beans: A Component Architecture for Java*. Also in URL: http://splash.javasoft.com/beans/WhitePaper.html

JavaSoft 1997a, *JavaBeans 1.0 Specification*, July 1997. Also in URL:http://splash.javasoft.com/beans

JavaSoft 1997b, *Remote Method Invocation Specification*. Also in URL: http://java.sun.com/products/rmi/index.html

Kohl et. al. 1989. *The Kerberos Network Authentication Service*. MIT Project Athena, 1989.

Kramer, D. 1996. *The Java Platform: A White Paper*, JavaSoft. Also in URL:http://java.sun.com/docs/white/platform/CreditsPage.doc.html

Kramer, M. 1996. *Distributed File Systems: IBM/Transarc DFS, Microsoft Distributed File System, Novell NetWare, Sun NFS*, 1996. White paper prepared for IBM Corp. Also in URL:http://www.transarc.com.

Linthicum, D. 1997. *The Java APIs*, Internet Systems, Vol. 10, No. 4. Also in URL:http://www.dbmsmag.com/9704i05.html

Object Management Group 1995. *The Common Object Request Broker: Architecture and Specification*, Revision 2.0. Also in URL: http://www.omg.org

Object Management Group 1997a. *A Discussion of the Object Management Architecture*. Also in URL: http://www.omg.org

Object Management Group 1997b. *CORBAservices: Common Object Services Specification*, March 1997. Also in URL: http://www.omg.org

Object Management Group 1997c, *The 1997 CORBA Buyers' Guide*, July 1997.

The Open Group 1996, *The Open Group Corporate Overview*, 1996.

The Open Group 1997, *The Open Group at a Glance*, 1997.

Orfali, R., Harkey, D., and Edwards, J., 1996a. *The Essential Client/Server Survival Guide*, 2nd Ed., Wiley & Sons, Inc.

Orfali, R., Harkey, D., and Edwards, J. 1996b. *The Essential Distributed Objects Survival Guide*, Wiley & Sons, Inc.

Orfali, R. and Harkey, D. 1997. *Client/Server Programming with Java and CORBA*, Wiley & Sons, Inc.

OSF 1991. *File Systems in a Distributed Computing Environment*, 1991.

Paepcke, A., et al. 1996. *Using Distributed Objects for Digital Library Interoperability*, IEEE Computer.

Rodriguez, L. 1997. *Java Beans: The Next Generation*, Java Developer's Journal, Vol. 1, No. 4, Also in URL: http://www.javadevelopersjournal.com/java/javajan/beans.htm

Ronayne, M., and Townsend, E. 1996. *Case Study: Distributed Object Technology at Wells Fargo Bank*, The Cushing Group, Inc. Also in URL: http://www.cushing.com

Rosenberry, W., Kenney, D., and Fisher, G. 1992. *Understanding DCE*, O'Reilly & Associates, Inc.

Roy, M., and Ewald, A. 1997. *Inside DCOM*, DBMS, Vol. 10, No. 4, pp. 27. Also in URL: http://www.dbmsmag.com

Rymer, J. 1996. *The Muddle in the Middle*, Byte Magazine, pp. 67 -- 70.

Shirley, J., Hu, W., and Magid, D. 1992. *Guide to Writing DCE Applications*, O'Reilly &Associates, Inc.

Umar, A. 1997. *Object-Oriented Client/Server Internet Environments*, Prentice Hall, Inc.

Vinoski, S. 1997. *CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments*, IEEE Communications, Vol. 14, No. 2.

Yang, Z., and Duddy, K. 1996. *CORBA: A Platform for Distributed Object Computing*, ACM Operating Systems Review, Vol. 30, No. 2, pp. 4 -- 31. Also in URL: http://www.dstc.edu.au/AU/research_news/omg/corba.html